Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

The original publication is available at http://dx.doi.org/10.1007/s10009-010-0145-y
Appeared in Software and Systems Modelling © Springer-Verlag

Rodin: An Open Toolset for Modelling and Reasoning in Event-B*

Jean-Raymond Abrial[1] and Michael Butler[2] and Stefan Hallerstede[3] and Thai Son Hoang[4] and Farhad Mehta[5] and Laurent Voisin[6]

[1] Independent Consultant, France
[2] University of Southampton, UK
[3] Heinrich-Heine-Universität Düsseldorf, Germany
[4] ETH Zürich, Switzerland
[5] Systransis Ltd, Switzerland
[6] Systerel, France

November 2009

Abstract. Event-B is a formal method for system-level modelling and analysis. Key features of Event-B are the use of set theory as a modelling notation, the use of refinement to represent systems at different abstraction levels and the use of mathematical proof to verify consistency between refinement levels. In this article we present the Rodin modelling tool that seamlessly integrates modelling and proving. We outline how the Event-B language was designed to facilitate proof and how the tool has been designed to support changes to models while minimising the impact of changes on existing proofs. We outline the important features of the prover architecture and explain how well-definedness is treated. The tool is extensible and configurable so that it can be adapted more easily to different application domains and development methods.

1 Introduction

We consider modelling of software systems and more generally of complex systems to be an important development phase. This is certainly the case in other engineering disciplines where models are often produced in the form of blueprints. We also believe that more complex models can only be written when the method of stepwise refinement is used. In other words, a model is built by successive enhancement of an original simple "sketch" carefully transforming it into more concrete representations. As an analogy, the first sketchy blueprint of an architect is gradually zoomed in order to eventually represent all the fine details of the intended building. On the way decisions are made concerning the way it can be constructed, thus yielding the final complete set of blueprints. We believe that formal notation is indispensable in such a modelling activity. It provides the foundation on which building models can be carried out, similar to the formal conventions that are used when drawing blueprints. Simply writing a formal text is insufficient, though, to achieve a model of high quality. We cannot test or execute a model to verify that the model has the properties that we demand of it. Similarly, we cannot open a window in the blueprint of a building. The only serious way to analyse a model is to reason about it, proving in a mathematically rigorous way that the properties are satisfied.

In order for formal modelling to be used safely and effectively in engineering practice, good tool support is necessary. Present day integrated development environments used for programming do carry out many tasks automatically in the background, e.g. [20], and provide fast feedback when changes are made to a program text. In particular, there is no need for the user to start processes like compilation. A program is written and then run or debugged without compiling it. In this paper we present the Rodin tool for Event-B [2] that applies these techniques used in programming to formal modelling. Instead of compilation, we are interested in proof obligation generation and automatically discharging trivial proof obligations. Instead of running a program we reason about models or analyse them.

The Event-B language and proof method are influenced by the B Method [1] and by Action Systems [8]. Following the Action System approach, system behaviour in Event-B is modelled by a collection of state variables and a collection of guarded actions that act on the state variables. This structure allows for modelling of highly concurrent systems. Following the B Method, the mathe-

* The continued development of the Rodin toolset is funded by the EU research project ICT 214158 DEPLOY (Industrial deployment of system engineering methods providing high dependability and productivity) www.deploy-project.eu. The toolset was originally developed as part of the project IST 511599 RODIN (Rigorous Open Development Environment for Complex Systems). The tool may be downloaded from www.event-b.org.

matical language for defining state structures and events is typed set theory. While the B Method is aimed at *software* development, Event-B is aimed at *system* development. When modelling a system, we may include facets of the non-software parts of the system and its environment, e.g., mechanical components. Systems that have been modelled in Event-B include a mechanical press, a train network, and a concurrent routing algorithm [2].

Verification by proof is not restricted to modelling. It has a long tradition in programming methodology, too, e.g. [27]. Software tools that support formal verification methods in programming have been developed, e.g. [11, 21]. We mention [11], in particular, because the Boogie architecture presented in the article provides characteristics similar to the Rodin tool. We quote two points from [11] about Boogie and present our view of them:

(1) "Design-Time Feedback". The tool is very responsive and provides almost immediate feedback that easily relates to the program (resp. model).

(2) "Distinct Proof Obligation Generation and Verification phases". This allows decoupling the development of the programming (resp. modelling) method and prover technologies. It also allows the origin of a proof obligation to be traced easily. This is particularly important when proofs fail.

The third point in the list describing Boogie in [11] is "Abstract Interpretation and Verification Condition Generation". The corresponding problem does not exist in the Event-B notation because it has been designed to be very close to the proof obligations by means of which we reason about Event-B. In Event-B all invariants are specified directly and do not need to be inferred by a tool. In Event-B refinement is used, that is, abstractions of a model do not need to be inferred but are specified and then refined. Technical difficulties encountered in Event-B stem more from the support of refinement and from the requirement that proof obligations appear transparent to the user. By transparency we mean that the user should look at the proof obligation as being part of the model. When a proof obligation cannot be proved, it should be almost obvious what needs to be changed in the model. When modelling, we usually do not simply represent some system in a formal notation. At the same time we learn what the system is and eliminate misunderstandings, inconsistencies, and specification gaps. In particular, in order to eliminate misunderstandings, we first must develop an understanding of the system. The situation is quite different when programming. When we start programming we should already understand what we are implementing. We do not look any longer at the system as a whole but only at the parts that we have to implement, and our main concern is doing this correctly. The task of a tool is to point out programming errors to the user.

The Rodin tool is intended to support construction and verification of Event-B models. The focus is very much on verifying models rather than on verifying programs. No assumptions are made about finiteness of structures and the main verification method is deductive proof; model checking can be used when structures are finite (see Section 10.1). Both automatic and interactive proof is provided (see Section 7). The main properties verified of models are well-definedness of expressions, invariant preservation and refinement between models. There are a number of ways in which Rodin provides design-time feedback in a responsive way. The user is encouraged to follow an incremental approach to modelling whereby verification is being applied automatically in the background as the model is being constructed. This means that each incremental change to the model represents a relatively small change to the set of proof obligations. By designing the proof method and tool to support incremental proof (see Section 5), the tools can cope with the verification required for each model increment in a timely fashion (of the order of a small number of seconds). Occasionally, a major re-factoring of models may be required and the re-proof effort will take more time.

We believe that the Rodin tool is a significant contribution to the goal of supporting modelling and proof in industrial settings. The contribution of this paper is to outline the principle functionality and design of the tool and to explain the rationale for the various design decision in the development of the tool.

The outline of the paper is as follows. We present a brief description of existing modelling and proof methods and tools in Section 2. After an introduction to the Event-B language in Section 3, we sketch the construction and proof of a simple Event-B model in Section 4 to help the reader understand the various functions of the tool. We then present the tool in more detail starting with the definition of the standard proof obligations in Section 5 followed by a description of the tool chain and tool interface in Section 6. The management of proofs and treatment of well-definedness are explained in Sections 7 and 8. Section 9 outlines how the tool is implemented on top of Eclipse. Before concluding we describe the extension possibilites of Rodin and some planned future developments.

An earlier paper [3] outlines the main features of Rodin. This current paper explains the key features of the tool in more depth, especially concerning the proof obligations, the proof manager, and the treatment of well-definedness.

2 Existing Tools for Modelling and Proof

We review a selection of formal modelling tools. It is not intended to be complete but to explain the kind of problems that we try to overcome with the Rodin tool.

The use of general purpose theorem provers with modelling notations like Z [16,39], Action Systems [6, 29], or Abstract State Machines [10,15] usually requires

a lot of expert knowledge in order to make efficient use of them when reasoning about formal models. This is not a problem of bad design of the theorem prover, but more a problem of bridging the gap between the notation and the logic underlying the theorem prover. General purpose theorem provers are well-suited to proving mathematical theorems in mathematical domains. The main problem solved by the theorem prover is to provide efficient ways to prove theorems. They are not specifically geared for modelling or the typical proof obligations associated with modelling. Theorem provers do assume that the problems to be proved, i.e. the proof obligations, are stated by the user and their proofs as such matter to the user. However, if the main interest of the user is modelling, the user is more concerned with understanding and learning about a model than with the proofs. In particular, generation of the proof obligations should be built into the tool to free the user from tedious work of writing them explicitly. In addition, we expect such a tool to be extensible and adaptable to cope with new and changing applications. This is not an issue with a general purpose theorem prover because proof obligation generation is manual anyway. In the Rodin tool we ensure that proof obligation generation remains extensible and adaptable.

Isabelle [36, 40] has been used with Z [16]. Although well-integrated the main problem remains that the user must explicitly specify proof obligations and is responsible for maintaining them. Another problem is that the user must understand the Isabelle logic as well as that of Z. To some degree this is alleviated by the Isar language [35] that extends Isabelle with more legible proofs. Similarly, abstract state machines (ASM) have been used with the KIV theorem prover [10]. The refinement theory used with ASM is stated in KIV and the user has to state the relevant theorems (proof obligations). When dealing with large models the amount of proof obligations is simply to high to load the user with this task [9]. Our tool overcomes these problems by maintaining proof obligations and by providing a prover that is tailored for first-order logic and set theory (which are the basic mathematical theories of Event-B). In the design of the tool, great care has been taken to easily relate proof obligations to a model, so that the user can quickly return to the model when a proof fails. The prover interface has also been designed to appear as natural as possible to the user. It gives a graphical representation of a sequent calculus for classical logic that has been further developed from the Click'n'Prove tool [4]. The major shortcoming of Click'n'Prove is that it is built on top of a theorem prover that executes proof scripts. As a consequence, feedback to the user is slow. In addition, the user must explicitly start tools to type-check a model, or generate proof obligations for it. Because the proof obligation generator has been developed for models of sequential programs with the B-Method [1], some proof obligations have variables renamed or are rewritten to a point where they are difficult to relate to the model. This violates our requirement for transparency. Following the experience with Click'n'Prove, we have also simplified Event-B (see Section 5) so that it does not hinder the design of a transparent proof obligation generator. In the Event-B tool, models are stored in a repository and manipulated like spread sheets; instead of storing a model as an abstract syntax tree and manipulating it as such, models are stored and manipulated in tables. Furthermore, all elements of a model (e.g. invariants, axioms) are named. This makes it possible for the tool to analyse models differentially, only generating proof obligations when necessary. The proof obligations are connected to the model by referring to involved repository elements.

The Z/EVES system [37] has a graphical front-end for Z specifications. It has automatic support for type-checking and some related properties. Although its prover is part of the tool, the user is responsible for stating relevant proof obligations. Z/EVES mostly provides a good interface for entering models graphically but less so for reasoning about them.

The approach of embedding a modelling notation into a general purpose theorem prover [16] like Isabelle [36] or Coq [14] provides a strong logical foundation. This is very satisfactory from a logicians point of view. From an industrial point of view, logical soundness is only one design consideration. We also need reactivity, i.e., immediate feedback, speed, and a notation and logic that is familiar to the user of the tool. This is very difficult to achieve in embedded designs. In the area of safety-critical embedded software, the approach of directly implementing provers has been proved fruitful. The Atelier B tool [18] has been used in large scale industrial projects, e.g. [9].

3 The Event-B Language

Event-B is defined in terms of a few simple concepts that describe a discrete event system and proof obligations that permit verification of properties of the event system. The syntax of Event-B is not fixed in order to allow for easy extension (e.g., introducing probabilities [34]). However, we present the notation using some syntactical conventions. The keywords **when**, **then**, **end**, and so on, are just delimiters to make the textual representation more readable.

An Event-B *model* consists of *contexts* and *machines*. In this description we focus on machines. A fuller description of Event-B can be found in [5].

Contexts contain the static parts of a model. These are *constants* and *axioms* that describe the properties of these constants.

Machines contain the dynamic parts of a model. A machine is made of a *state*, which is defined by means of *variables*. Variables, like constants, correspond to simple mathematical objects: sets, binary relations, functions,

numbers, etc. They are constrained by *invariants* $I(v)$ where v are the variables of the machine. Invariants are supposed to hold whenever variable values change. But this must be proved first (see Section 5.2).

Besides its state, a machine contains a number of *events* which specify how the state may evolve. Each event is composed of a *guard* and an *action*. The guard is the necessary condition under which the event may occur. The action, as its name indicates, determines the way in which the state variables are going to evolve when the event occurs. An event may have *parameters* that are local to that event. Parameters can serve different functions, for instance, to model arrays of events or as communication channels in composition of machines [17].

An event may be executed only when its guard holds. Events are *atomic* and when the guards of several events hold simultaneously, then *at most one of them* may be executed at any one moment. The choice of event to be executed is non-deterministic. An event, named evt, is specified in one of the three following forms:

$$\begin{aligned}
&\mathsf{evt} \mathrel{\widehat{=}} \textbf{any } t \textbf{ where } P(t,v) \textbf{ then } S(t,v) \textbf{ end}\\
&\mathsf{evt} \mathrel{\widehat{=}} \textbf{when } P(v) \textbf{ then } S(v) \textbf{ end}\\
&\mathsf{evt} \mathrel{\widehat{=}} \textbf{begin } S(v) \textbf{ end} \quad ,
\end{aligned}$$

where $P(\ldots)$ is a predicate denoting the guard, t denotes parameters that are local to the event, and $S(\ldots)$ denotes the action that updates some variables. The variables of the machine containing the event are denoted by v. The first event form is the most general one where an event has some parameters t and a guard $P(t,v)$. It can be executed in a state represented by v provided $P(t,v)$ holds for some t; its effect on v is specified by action $S(t,v)$. The second event form is used if an event does not have any parameters. The third form is used if an event does not have parameters and its guard is true.

An action consists of a collection of *assignments* that modify the state simultaneously. An assignment has one of the following three forms:

Assignment		**Before-After Predicate**
$x := E(t,v)$	–	$x' = E(t,v)$
$x :\in E(t,v)$	–	$x' \in E(t,v)$
$x :\mid Q(t,v,x')$	–	$Q(t,v,x')$,

where x are some variables, $E(\ldots)$ denotes an expression, and $Q(\ldots)$ a predicate. Assignments of the form $x := E(t,v)$ are called *deterministic*, the other two forms are called *nondeterministic*. Form $x :\in E(t,v)$ assigns x to an element of a set, and form $x :\mid Q(t,v,x')$ assigns to x a value satisfying a predicate. Simultaneity of a collection of assignments is expressed by conjoining the before-after predicates of an action. Variables y that do not appear on the left hand side of an assignment of action do not change. Formally this is achieved by conjoining $y' = y$ to the before-after predicate of the action.

In order to be able to provide better tool support, invariants, guards, actions are lists of named predicates and assignments. These names can be used to refer to these objects from within the documentation of a machine. But foremost, these names are used to identify all objects and provide helpful information about the origin of proof obligations in the prover interface. The different predicates in the list are implicitly conjoined.

The mathematical language of Event-B has a simple type system. The types are *basic types* (such as integers or given sets that are specific to a model) or are formed from the *cartesian product* and *powerset* type constructors. Structures such as relations and functions are defined by combining these type constructors. A type inference system is used to infer types of constants, variables and event parameters from axioms, invariants and guards.

4 Incremental construction of an example

In this section we outline the construction of a small Event-B model and its refinement using the Rodin tool. Our aim is to illustrate the interaction between modelling and proof during model construction. This will serve to motivate the reactive nature of the support provided by the Rodin tool as models are constructed incrementally. Although we present this example before presenting the details of the proof method (Section 5), it is sufficiently self-explantory for the general reader at this stage. The proof obligations being verified for the example are invariant preservation, refinement and well-definedness. We assume knowledge of basic set theory.

The model is of a system for checking registered users in and out of a building. We start the construction of the model by dealing only with registration of users. In the tool we create a new context and introduce a given set $USER$ in the context. We create a new machine and add a variable *register* to the machine to represent the set of registered users. We create an invariant to specify the register as a set of users:

$$\mathsf{inv1} \quad register \subseteq USER$$

The type inference mechanism infers the type of the *register* variable, i.e., $\mathbb{P}(USER)$, from this invariant.

We create an event to add a new user to the register:

$$\begin{aligned}
&\mathsf{Register} \mathrel{\widehat{=}} \textbf{any } u \textbf{ where}\\
&\qquad \mathsf{grd1} \quad u \in USER \setminus register\\
&\textbf{then}\\
&\qquad \mathsf{act1} \quad register := register \cup \{u\}\\
&\textbf{end}
\end{aligned}$$

Here, parameter u represents the identity of the new user. At this stage in the formal development, we do not consider whether u is an input or an output of the system rather we simply state that the new user is not already in *register* (grd1).

With the above elements (set $USER$, variable *register*, invariant inv1 and event *Register*) added to the project, the only error message we get is that the *register* variable has not been initialised. This is remedied by adding the action $register := \varnothing$ to the machine initialisation. At this stage the model results in no proof obligations since the invariant inv1 is nothing stronger than a typing constraint.

Now we add variables to represent the set of people who are in the building (*in*) and those that are outside the building (*out*). These are typed and constrained to be subsets of *register* through the following invariants:

$$\mathsf{inv2} \quad in \subseteq register$$
$$\mathsf{inv3} \quad out \subseteq register$$

Note that while these invariants allow the type inference mechanism to infer the types of *in* and *out*, they are stronger that typing invariants since *register* is a variable and not a type. We ensure that *in* and *out* are initialised to be empty. We have an obvious requirement that a user cannot be simultaneously inside and outside the building so we add a further invariant:

$$\mathsf{inv4} \quad in \cap out = \varnothing$$

The resulting model now gives rise to 6 proof obligations in total; 3 of these are to verify that the initialisation establishes invariants inv2 to inv4 and 3 are to verify that the *register* event maintains invariants inv2 to inv4. All 6 proof obligations are discharged automatically. The general definition of the proof obligations is explained in Section 5.

We add events to model users entering and leaving the building. Our first attempt at the *Enter* event is

$$\begin{array}{l} \mathsf{Enter} \mathrel{\widehat{=}} \textbf{any } u \textbf{ where} \\ \quad \mathsf{grd1} \quad u \in out \\ \textbf{then} \\ \quad \mathsf{act1} \quad in := in \cup \{u\} \\ \textbf{end} \end{array}$$

This event gives rise to 3 new proof obligations (1 for each of *inv2* to *inv4*), 1 of which is not automatically discharged. Using the proof obligation explorer we can inspect this unproved proof obligation and see that it has hypotheses and a goal as follows:

$$\begin{array}{ll} \mathsf{Hyp1:} & in \cap out = \varnothing \\ \mathsf{Hyp2:} & u \in out \\ \vdash & \\ \mathsf{Goal:} & (in \cup \{u\}) \cap out = \varnothing \end{array} \quad (1)$$

Clearly this cannot be proved: if $u \in out$ then $\{u\} \cap out$ is not empty. Thus either the invariant it is associated with (inv4) is wrong or the *Enter* event is wrong and one or both need to be changed. The obligation explorer provides hyperlinks to both inv4 and *Enter* to facilitate any changes to either. In this case we decide that the error is in the *Enter* operation since we neglected to remove the user from the variable *out*. We remedy this by clicking on the link to the *Enter* event and adding the following action to this event:

$$\mathsf{act2} \quad out := out \setminus \{u\}$$

This addition results in all proof obligations being discharged automatically. Note that having a proof obligation that is not automatically discharged does not necessarily mean there is an error in the model. It may be due to a limitation of the automatic prover and instead the obligation may be provable using the interactive prover. The interactive prover of Rodin is explained in more detail in Section 6.

A further requirement on the model is that each registered user must either be inside or outside the building. Our existing invariants are not sufficient to express this property so we add a further invariant:

$$\mathsf{inv5} \quad register \subseteq in \cup out$$

This addition gives rise to 3 new proof obligations, 1 of which is not automatically discharged:

$$\begin{array}{ll} \mathsf{Hyp1:} & register \subseteq in \cup out \\ \mathsf{Hyp2:} & u \in USER \setminus register \\ \vdash & \\ \mathsf{Goal:} & (register \cup \{u\}) \subseteq in \cup out \end{array} \quad (2)$$

Clearly this obligation is not provable: if u is not in *register*, then it is not in $in \cup out$. The obligation explorer tells us that this proof obligation arises from both inv5 and the *Register* event. Inspection of the *Register* event shows that it adds a user u to *register* but not to either *in* or *out*. We remedy this by deciding that newly registered users should be recorded as being outside the building and adding the following action to the existing *Register* event:

$$\mathsf{act2} \quad out := out \cup \{u\}$$

All proof obligations of the resulting model are automatically discharged.

We now outline a data refinement of this model. Let us assume that we decide to implement this model as a simple database and replace the two abstract variables *in* and *out* with a single *status* function. The variables of the refined model (the concrete variables) are *register*, as before, and a new variable *status*, a total function from *register* to $STATUS$:

$$\mathsf{inv6} \quad status \in register \to STATUS$$

$STATUS$ is an emumerated type with distinct values IN and OUT.

The abstract *Enter* event is guarded by the condition that $u \in out$. In the refined *Enter* event, this guard is replaced by a condition on the *status* function. The

refined event updates the *status* function rather than modifying the *in* and *out* variables:

$$\begin{array}{l}\mathsf{Enter} \mathrel{\widehat{=}} \textbf{refines } \mathsf{Enter}\\ \quad \textbf{any } u \textbf{ where}\\ \quad\quad \mathsf{grd1} \quad u \in register\\ \quad\quad \mathsf{grd2} \quad status(u) = OUT\\ \quad \textbf{then}\\ \quad\quad \mathsf{act1} \quad status(u) := IN\\ \quad \textbf{end}\end{array}$$

The clause **refines** Enter indicates that the refined *Enter* event refines the *Enter* event of the abstract machine. In general, the names of refined events may differ from the corresponding abstract event so that a refined event must include an explicit reference to some event of the abstract machine. This refined event gives rise to an unproved refinement proof obligation as follows:

$$\begin{array}{ll}\mathsf{Hyp1}: & u \in register\\ \mathsf{Hyp2}: & status(u) = OUT\\ \vdash & \\ \mathsf{Goal}: & u \in out\end{array} \qquad (3)$$

This proof obligation arises because of the need to show that the guard of a refined event implies the corresponding abstract guard. We can see that the hypotheses come from the guards of the refined event (grd1 and grd2) while the goal is the guard of the abstract event. As it stands the goal cannot be proven since we have not stated any invariant relating the concrete *status* variable and the abstract *in* and *out* variables. Such an invariant is called a *gluing invariant* and will be described more precisely in the next section. Refinement proofs rely on such gluing invariants. We could simply convert the above proof obligation (3) into a gluing invariant as follows:

$$\begin{array}{ll}\mathsf{inv7} & \forall u \cdot u \in register \ \wedge \ status(u) = OUT\\ & \Rightarrow \quad u \in out\end{array}$$

Intuitively this invariant is reasonable since it is states that for any registered user whose status is OUT at the concrete level, that user is in the set *out* at the abstract level. Adding this invariant allows proof obligation (3) to be discharged automatically. A similar invariant about users in the set *in* can be used to discharge a proof obligation for a refined *Leave* operation. Note that invariant inv7 refers to variables of both the abstract and refined machines. Invariants of a refined machine may refer to variables of the abstract machine. These are so-called *gluing invariants* and are explained further in Section 5.

We consider one other proof obligation associated with the refined *Enter* event:

$$\begin{array}{ll}\mathsf{Hyp1}: & u \in register\\ \vdash & \\ \mathsf{Goal}: & u \in dom(status) \wedge\\ & status \in USER \pfun STATUS\end{array} \qquad (4)$$

This is a well-definedness obligation associated with the expression $status(u)$ in guard grd2 of the refined *Enter* event. Function application in Event-B gives rise to a well-definedness obligation which requires that argument u is in the domain of *status* and that *status* is a *partial* function (and not just a relation). Because of the declaration of *status* (inv6), obligation (4) is discharged automatically. The well-definedness obligations are explained in more detail in Section 8.

We have now completed our construction of the small Event-B model and its refinement. With the old style tools for B, after constructing the model, we would have separately invoked the proof obligation generator and then the automatic prover. With the Rodin tool, this is taken care of automatically as we construct the model. Based on undertaking a range of Event-B developments, large and small, with Rodin, our experience is that by making use of the feedback from the tool as we construct the model, e.g., the unproved proof obligations, we are guided towards construction of a model that has less errors and is more easily proved than if we were to delay any proof analysis until after constructing the full model.

5 The Event-B Proof Method

In this section we outline standard proof obligations associated with Event-B models.

5.1 Feasibility of Assignment

Recall from Section 3 that a nondeterministic assignment has the following form:

$$x :| Q(t, v, x').$$

Event-B requires actions to be feasible under the guard of the corresponding events, that is, when its guard is true the action of an event must yield a successor state. For the non-deterministic assignment we must prove

$$\begin{array}{l}I(v)\\ P(t, v)\\ \vdash\\ (\exists\, x' \cdot Q(t, v, x'))\end{array} \quad ,$$

where $I(v)$ is the invariant of the machine and $P(t, v)$ the guard of the event.

5.2 Consistency of a Machine

Once a machine has been written, one must prove that it is *consistent*. This is done by proving that each event of the machine preserves the invariant. More precisely, it must be proved that the action associated with each event modifies the state variables in such a way that the

modified variables satisfy the invariant, under the hypothesis that the invariant holds presently and the guard of the event is true. For a machine with state variable v, invariant $I(v)$, and an event **when** $P(v)$ **then** $v := E(v)$ **end** the statement to be proved is the following:

$$\begin{array}{l} I(v) \\ P(v) \\ \vdash \\ I(E(v)) \quad . \end{array} \tag{5}$$

Note that, in practice we carry out a decomposition of (5) according to the lists of named invariants, guards, and actions. So statement (5) is not the proof obligation the user gets to see. Instead the user sees a collection of simpler proof obligations.

Inspection of (5) reveals the simplicity that is at the core of the Event-B method. In order to arrive at statement (5) we simply copy elements from the model and apply some basic rewriting. For the consistency proof obligation we copy the invariant $I(v)$ the guard $P(v)$ of the event in the hypothesis of the proof obligation and the modified invariant $I(E(v))$ where v has been replaced by $E(v)$ in the goal. This makes it easy to relate elements of the model to corresponding proof obligations when using Event-B in practice which is important when making incremental changes to a model as shown in Section 4. For example, proof obligation (1) is a consistency obligation that comes from act1 of the abstract *Enter* event and inv4.

5.3 Refining a Machine

Machine refinement provides a means to introduce more detail about the dynamic properties of a model [5]. The theory of refinement is a simplified form of the corresponding notion of the Action Systems formalism [8] that has inspired the development of Event-B. Action Systems and and other refinement theories support both forward and backward refinement. In common with the B-Method, Event-B refinement currently supports forward refinement; backwards refinement is not currently supported.

Refining a machine consists of refining its state and its events. A concrete machine (with regards to the more abstract one) has a state that should be related to that of the abstraction by a so-called *gluing invariant*, which is expressed in terms of a predicate $J(v, w)$ connecting the abstract state represented by the variables v and the concrete state represented by the variables w. We introduce first refinement proof obligations for events without parameters to illustrate the principle. Afterwards, we show how we deal with parameters using *witnesses*. We deal with non-deterministic assignments similarly as explained in [24].

Each event of the abstract machine is refined to one or more corresponding events of the concrete one. Informally speaking, a concrete event is said to refine its abstraction (1) when the guard of the former is stronger than that of the latter (guard strengthening), and (2) when the gluing invariant is preserved by the conjoined action of both events. In the case of an abstract event *abs* and a corresponding concrete event *con* of the form

$$\begin{array}{l} \mathsf{abs} \mathrel{\widehat{=}} \textbf{when } P(v) \textbf{ then } v := E(v) \textbf{ end} \\ \mathsf{con} \mathrel{\widehat{=}} \textbf{when } Q(w) \textbf{ then } w := F(w) \textbf{ end} \quad , \end{array}$$

the statement to prove is the following:

$$\begin{array}{l} I(v) \\ J(v, w) \\ Q(w) \\ \vdash \\ P(v) \ \wedge \ J(E(v), F(w)) \quad , \end{array} \tag{6}$$

where $I(v)$ is the abstract invariant and $J(v, w)$ is the gluing invariant. Similarly to (5) the user never gets to see (6) but only the decomposed form.

In the case of events *abs* and *con* with parameters

$\mathsf{abs} \mathrel{\widehat{=}}$ **any**	$\mathsf{con} \mathrel{\widehat{=}}$ **any**
t	u
where	**with**
$P(t, v)$	$t = W(u, w)$
then	**where**
$v := E(t, v)$	$Q(u, w)$
end	**then**
	$w := F(u, w)$
	end ,

we have to prove:

$$\begin{array}{l} I(v) \\ J(v, w) \\ Q(u, w) \\ \vdash \\ P(W(u, w), v) \ \wedge \ J(E(W(u, w), v), F(u, w)) \quad , \end{array} \tag{7}$$

where $W(u, w)$ are called *witnesses*; see [24]. Witnesses are specified in the model because they provide an essential insight into the refinement relationship of the abstract and the concrete event. (If a variable or parameter is repeated in a refinement, it is assumed that the concrete one and the abstract one are identical.) One could say, they provide a local gluing invariant. They also permit decomposition of (7) similarly to statement (5) in Section 5.2. Without the use of witnesses, the goal would be preceded by an existential quantifier $\exists\, t$ and the witness $W(u, w)$ would have to be provided interactively by the user during the proof of the obligation. By making the witness an explicit part of the model, its definition is more obvious to the modeller and refinement proofs go through more automatically.

5.4 Adding New Events in a Refinement

When refining a machine by another one, it is possible to *add new events*. Such events must be proved to refine a dummy event that does nothing (skip) in the abstraction. Moreover, it may be proved that the new events cannot collectively take control forever. For this, a unique *variant expression* $V(w)$ has to be provided, that is decreased by each new event. We refer to this as a *convergence* proof obligation.

In case the new event has the form:

$$\mathsf{evt} \mathrel{\widehat{=}} \textbf{when } R(w) \textbf{ then } w := G(w) \textbf{ end} \quad ,$$

the following statements (8) and (9) have to be proved:

$$\begin{array}{l} I(v) \\ J(v,w) \\ \vdash \\ J(v, G(w)) \end{array} \tag{8}$$

$$\begin{array}{l} I(v) \\ J(v,w) \\ \vdash \\ V(w) \in \mathbb{N} \ \wedge \ V(G(w)) < V(w) \quad , \end{array} \tag{9}$$

where we assume that the variant expression is a natural number (but it can be more elaborate).

5.5 Event extension

A very common form of refinement is called *superposition* refinement [6] where only new elements are added to a machine and none of the existing variables or parameters is *data-refined* [7]. In this case we only need to specify what is new in each refined event: new parameters, new guards, new actions. By doing this, making changes to a model becomes very efficient which is particularly important in conjunction with the incremental approach to modelling promoted by the Rodin tool.

In practice changes to a model occur as often to abstract machines as to refinements. Suppose, we have a model with 10 superposition refinements where we were to repeat the contents of all abstractions in each refinement. If we needed to change a guard of an event at the most abstract machine in that model it would be necessary to carry out the same change in all 9 refinements. The use of event extension means we only have to make a change in one place.

Let *abs* be an abstract event and *con* be an extension of *abs*:

$$\begin{array}{l} \mathsf{abs} \mathrel{\widehat{=}} \textbf{when } P(v) \textbf{ then } v := E(v) \textbf{ end} \\ \mathsf{con} \mathrel{\widehat{=}} \textbf{when } Q(w) \textbf{ then } w := F(w) \textbf{ end} \quad . \end{array}$$

Then the contents of event *abs* is automatically replicated in event *con*:

$$\begin{array}{l} \mathsf{con} = \textbf{when} \\ \qquad P(v) \ \wedge \ Q(w) \\ \quad \textbf{then} \\ \qquad v, w := E(v), F(w) \\ \quad \textbf{end} \quad . \end{array}$$

The Rodin tool takes care of the replication and avoids generation of refinement proof obligations associated with extended events.

5.6 Modelling language support for proof

In the description of Event-B above at some points we have referred to the simplicity of Event-B, in particular, with respect to its support for proof as its main technique for reasoning. As a matter of fact, during the design of Event-B much attention has been paid to this. The entire notation has been designed to facilitate simple proof obligation generation and efficient retrieval of old proofs associated with proof obligations.

Simple proof obligation generation is achieved by the reduced structure of the notation. Contexts, machines, and events provide only the structure necessary to allow the reasoning outlined above. All proof obligations are generated from the model with as little rewriting as possible. This is done in order to permit the user of Rodin easy switching between modelling and proving. It is essential that the user recognises immediately the elements of a model that make up a proof obligation, and conversely that the user can easily imagine the proof obligations associated with elements being modified.

Efficient retrieval of proof obligations is achieved by naming proof obligations systematically using labels associated with each element of a model. As an illustrative example we consider a model with two invariants

$$\begin{array}{ll} \mathsf{inv1} & I_1(v) \\ \mathsf{inv2} & I_2(v) \end{array}$$

and one event

$$\begin{array}{l} \mathsf{Evt} \mathrel{\widehat{=}} \textbf{any } t \textbf{ where} \\ \qquad \mathsf{grd1} \quad P_1(t,v) \\ \quad \textbf{then} \\ \qquad \mathsf{act1} \quad v := E_1(t,v) \\ \quad \textbf{end} \quad . \end{array}$$

From this model two proof obligations

$$\begin{array}{ll} \mathsf{Evt/inv1}: & \mathsf{Evt/inv2}: \\ I_1(v) & I_1(v) \\ I_2(v) & I_2(v) \\ P_1(t,v) & P_1(t,v) \\ \vdash & \vdash \\ I_1(E_1(t,v)) & I_2(E_1(t,v)) \end{array}$$

are generated. Note, how easy it is to match model elements and proof obligations and how this is reflected in the naming. The naming remains when the elements I_1, I_2, P_1, or E_1 are changed. Thus, it is trivial for the Rodin tool to locate proofs associated with proof obligations before the change. The speed of this is crucial for the incremental modelling approach to work. In a complex model there are usually many proof obligations but the feedback provided by the tool should depend as little as possible on the number of proof obligations. The development of the concept of witnesses also started with efficiency considerations. Without witnesses the goal of refinement proof obligations would contain a conjunction enclosed by an existential quantifier rendering decomposition of the goal according to the labelled conjuncts impossible. By the use of witnesses the existential quantifier in the goal disappears [24].

The systematic naming scheme also contributes to simplicity in the sense that the user can easily locate proof obligations when analysing specific elements of a model.

Which proof obligations are to be generated for a model is controlled by attributes associated with events. Proof obligations for convergence are associated with events by providing them with a corresponding "convergence attribute". Similarly, event extension is available by attributing events correspondingly.

5.7 Differential Proving

In Event-B changes to a model are expected to occur frequently. The user is expected to improve a model in small increments. Changes happen for various reasons. Most often a model is changed because of increased understanding that has been gained through the modelling and reasoning. Sometimes a model is changed simply because of small mistakes that occur when typing formal text. And sometimes changing a model facilitates proof. When a model changes, the impact on proofs already carried out should be as small as possible. Obviously, the user should not be asked to redo a valid proof. But the same is expected concerning the tool. Proving is very time consuming and should be avoided in order to achieve better reactivity of Rodin and in order to encourage incremental modelling. We want the user to make frequent changes. Hence, the tool should manage proofs differentially only redoing a proof when its necessary. We use a number of techniques to achieve this.

Proof obligations are filtered along three stages. The first stage is purely syntactic. If a newly generated proof obligation is syntactically identical to the old proof obligation, the prover assumes validity of the old proof for the new proof obligation. This process is speeded-up by the naming scheme that permits fast retrieval of proof obligations and proofs. In the second stage, proofs of proof obligations that have changed syntactically are analysed. For each proof the tool records the hypotheses used for the proof to succeed. If none of the used hypotheses has changed and the goal has not changed, the proof is assumed valid for the new proof obligation. Finally, in the third stage, the proof is replayed attempting to rename identifiers that have been freed in quantified expressions during the proof. If this fails, the old proof has to be carried out again in full.

6 Tool Chain and Tool Interface

The software tool support for Event-B should not be just another theorem prover. It should be a modelling tool that constrains modelling activity as little as possible. Powerful theorem provers are available [14, 19, 26, 36] but not enough attention has been paid in formal methods to tool support for the modelling activity per se. Traditionally, it is assumed that one begins a formal development with a specification and develops it into a correct implementation. The flaw in this description is that, initially, there is no specification. Writing a specification involves making errors. The Rodin tool takes this into account by being reactive and efficiently supporting incremental changes to models. Development towards an implementation will profit from this, too. In fact, we consider both, writing a specification and implementing it, to be part of the modelling activity.

6.1 Tool Chain

The Rodin tool chain consists of three major components: the static checker (SC), the proof obligation generator (POG), and the proof obligation manager (POM). Their connection is shown in Figure 1 and their purpose is described below.

The static checker (SC) for Event-B analyses Event-B contexts and Event-B machines and provides feedback to the user about syntactical and typing errors in them.

The proof obligation generator (POG) for Event-B generates proof obligations many of which have been outlined in Section 5. These cover proof obligations for

- feasibility
- event consistency
- refinement
- convergence
- well-definedness (see Section 8)

The POG has been designed to fit the requirement of responsiveness needed for an incremental modelling approach. As a result proof obligation generation must be very fast. This led to the decision that the POG only performs "mild" rewriting of modelling elements (see Section 5.6) and does not attempt any proof. The POG tracks changes and tries to stop as soon as possible; for instance, if only an event is changed in a machine, only the proof obligations for that machine and

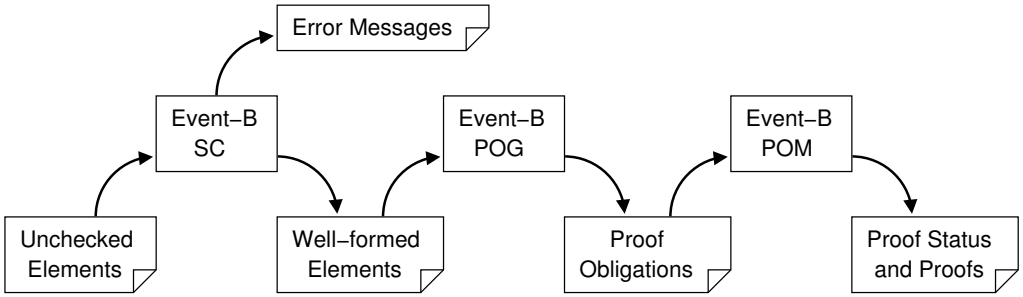


Fig. 1: Tool-Chain in Event-B Core

its immediate refinement need to be generated. Proof obligations are always computed for entire machines because it would take more time to determine which proof obligations need to be generated than generate all proof obligations and check afterwards which proof obligations have changed.

The proof obligation manager (POM) keeps track of proof obligations and associated proofs and is described in Section 7.

6.2 On the Role of the Static Checker

The static checker has two main objectives:

(1) to generate feedback to the user;
(2) to filter elements of a model that are not well-formed.

Concerning (1), the design decision has been made that only the SC generates error messages (whereas the POG does not). This corresponds roughly to the traditional splitting of compilers into parsers and code generators. This architecture benefits the responsiveness of the tool in that the POG needs only be started if the output of the static checker has changed.

Concerning (2), the design decision has been made that the SC does not reject whole machines or contexts but only elements thereof that are not well-formed. Well-formedness is defined in terms of the syntax of the mathematical language, dependencies between modelling elements, and type-correctness of all formulas and declared identifiers. The POG does not check the output of the SC. It just generates proof obligations from it. In this sense one could say that the SC carries out a "differential verification" of the precondition of the POG. As a consequence of the well-formedness of its input the POG does not generate proof obligations that just correspond to type-checking identifiers.

6.3 The Graphical User Interface

The graphical user interface consists of two parts: one user interface for modelling (MUI) and one user interface for proving (PUI). Figure 2 shows how the core components and the user interface are integrated. The proving user interface does not access proof obligations and proofs directly but uses the services of the proof obligation manager. Figure 3a and 3b show screen shots of the modelling perspective and the proving perspective respectively.

The two user interfaces are connected by the tool chain of the Event-B core. They are available to the user in form of Eclipse perspectives between which the user can switch easily. The two perspectives are seamlessly integrated so that it is not suggested that modelling and proving are different activities. The user is intended to perceive reasoning about models as being part of modelling. Proof obligations are equipped with hypertext links so that the user can select instantaneously modelling elements related to that proof obligation.

7 Proof Obligation Manager

7.1 Overview architecture

The task of the Proof Obligation Manager is to maintain the proof status (e.g., discharged/reviewed/pending) and the proofs associated with the obligations. Hence the Proof Obligation Manager needs to work both automatically (as a part of the tool-chain) and interactively (with the Proving UI). How the Proof Obligation Manager handles the synchronisation in different modes is described later in Section 7.6.

Internally, the Proof Obligation Manager architecture is separated into two parts: "extensible" and "static" part. The extensible part is responsible for generating individual proof rules (see Section 7.2) which is used for proving proof obligations (which are respresented as sequents). The static part is responsible for putting these proof rules together to construct and maintain proofs. The components that generate proof rules are called "reasoners" (Section 7.3).

The Proof Obligation Manager builds a (possibly partial) proof for a proof obligation by constructing "proof trees" (Section 7.4).

However, the users of the Proof Obligation Manager do not work directly with reasoners. In order to encapsulate frequently used proof construction and manipu-

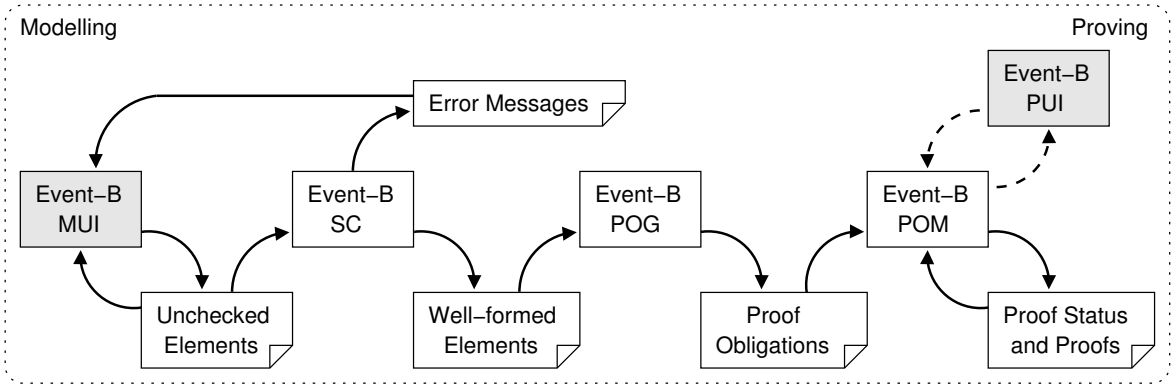
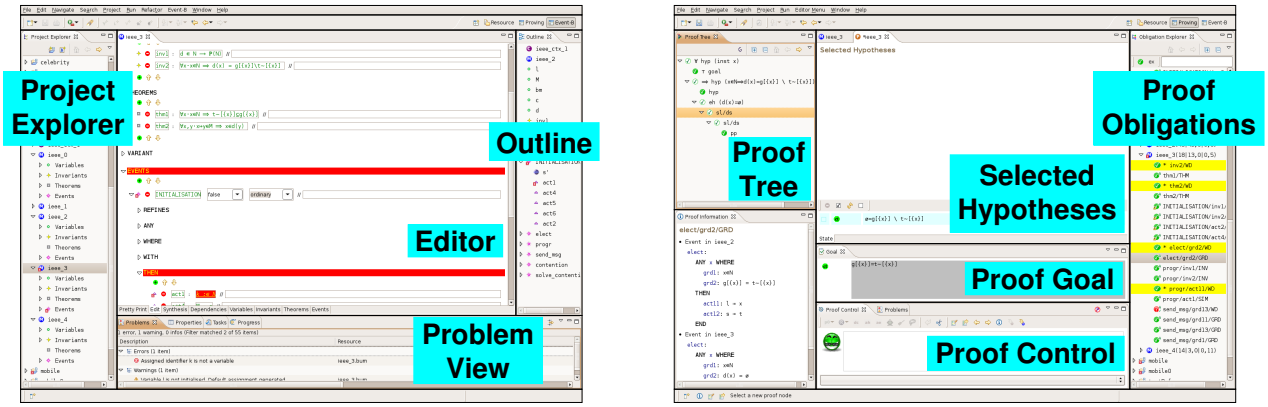


Fig. 2: The User Interface



(a) Modelling Perspective

(b) Proving Perspective

Fig. 3: Default Perspectives of Rodin

lation steps, the Proof Obligation Manager provides the concept of "tactics" (Section 7.5).

The Proof Obligation Manager can be extended by adding new tactics and reasoners.

7.2 Proof Rules

In its pure mathematical form, a proof rule is a tool to perform formal proof and is denoted by:

$$\frac{A}{C}$$

where A is a (possibly empty) list of sequents: the antecedents of the proof rule; and C is a sequent: the consequent of the rule. We interpret the above proof rule as follows: the proofs of *each sequent* of A together give a proof of sequent C.

In Rodin, the proof rule has more structure in order to reduce the storage space and more importantly, to support proof reuse. In its simplest form, the proof schema in Rodin is as follows:

$$\frac{H, H_{A_0} \vdash G_{A_0} \quad \ldots \quad H, H_{A_n} \vdash G_{A_n}}{H_v, H_u \vdash G_u}$$

Where:

- H_u is the set of *used hypotheses*
- H_v is the set of *unused hypotheses*
- G_u is the *used goal*, G_u may be absent if the proof rule does not depend on the goal.
- H_{A_i} is the set of added hypotheses corresponding to the ith antecedent.
- G_{A_i} is the new goal corresponding to the ith antecedent.
- $H = H_v \cup H_u$ is the set of all hypotheses of the consequent.

Note that H_v is a meta-variable that can be instantiated. Different instantiations will give rise to different proof rules. Additionally, an antecedent may contain a number of forward inferences for hypotheses that allows for calculating finer grained hypotheses dependencies. An example follow in Section 7.3.1 (**SIM**).

Given a proof rule of the form mentioned above, the proof rule is *not* applicable to a sequent if the goal of the sequent is not exactly the same as the used goal G_u (when G_u is present) or any of the required hypotheses in H_u is missing in the sequent. In the case of applicability, the output of the process of applying a proof rule to a sequent is a set of sequents corresponding to the antecedents. The required hypothesis are treated as the used hypotheses H_u and the Proof Obligation Manager instantiates H_v with the set of hypotheses of the input sequent not in H_u. More details on using this approach to represent proof rules can be found in [31].

7.3 Reasoners

Reasoners are responsible for generating proof rules. The input of a reasoner is a sequent and possibly some optional input (e.g. a predicate in the case of the Cut Rule). The reasoner is successful if it can generate a proof rule which is applicable to the input sequent. In this case, this proof rule is the output of the reasoner and is trusted by the Proof Obligation Manager. How the reasoners generate proof rules is not visible to the other parts of the Proof Obligation Manager. The only assumptions that the Proof Obligation Manager makes about the reasoner are as follows:

Logically Valid: A generated proof rule must be valid (i.e. can be derived) in the mathematical logic.

Re-playable: A reasoner must work deterministically, i.e. the reasoner must generate the same proof rule if given the same input.

7.3.1 Examples of Reasoners

The list of complete proof rules implemented in the Rodin platform including information on how they are used by default (i.e., automatic/manual) is available on-line at http://wiki.event-b.org/index.php/Inference_Rules. This section gives some examples of the available reasoners.

Simplifier: The simplifier derives new hypotheses and simplifies the goal of a sequent according to some predefined simple rewriting rules. Examples of rewriting rules are

$$\begin{array}{rcl} E = E & == & \top \\ \top \Rightarrow P & == & P \\ E + 0 & == & E \end{array}$$

The full set of rewriting rules is available on-line at http://wiki.event-b.org/index.php/All_Rewrite_Rules.

The simplifier generates proof rules of the following form:

$$\frac{H, H' \vdash G'}{H \vdash G}\ \textbf{SIM}$$

Here, H' and G' are the rewritten form of H and G. H' and G' are formed by iteratively applying any applicable rewriting rules until none are applicable. Application of the **SIM** inference rule results in the goal changing from G to the rewritten G' and the rewritten hypotheses H' being added. The original hypotheses H are maintained. For example, if we have the following sequent

$$a = a \Rightarrow b = c \ \vdash\ b + 0 = c\ ,$$

the **SIM** reasoner applied to this sequent generates the following inference rule:

$$\frac{a = a \Rightarrow b = c,\ b = c \ \vdash\ b = c}{a = a \Rightarrow b = c \ \vdash\ b + 0 = c}\ \textbf{SIM}$$

Goal in Hypotheses: This reasoner generates proof rules of the following form

$$\frac{}{H, G \vdash G}\ \textbf{Hyp}$$

The reasoner is successful if the goal of the input sequent appears in the set of hypotheses. Here both the hypothesis G and the goal G are used.

Split Conjunctive Goal: This reasoner generates proof rules of the following form

$$\frac{H \vdash P \qquad H \vdash Q}{H \vdash P \wedge Q}\ \textbf{ConjGoal}$$

The reasoner sets the used goal to be $P \wedge Q$. The used hypotheses will be inherited from the sub-goals and there are no additional used hypotheses.

7.4 Proof Trees

Proof trees are recursive structures based on *Proof Tree Nodes*. Each node has three components: a sequent, a proof rule (possibly *null*) and a list of children nodes (possibly *null*, when the proof rule is present or *empty* in the case when the node has no children). A proof tree corresponds to a proof obligation if the sequent of the root node of the tree is the same as the obligation.

A proof tree node is either *pending* if no rule is applied to this node or *non-pending* otherwise. A proof tree is valid if all its nodes are valid. The validity of a proof tree node is defined recursively as follows:

- For a pending node, its children must be *null*.
- For a non-pending node, its children must not be *null*. Moreover the proof rule is applicable to the *sequent* of the node and the children correspond to the result of the application of the rule to the sequent. Lastly, all of the child nodes are valid.

The Proof Obligation Manager provides the following operations on Proof Trees:

Construction: Create an initial proof tree corresponding to an input sequent.

Rule Application: A proof tree grows when a rule applies to one of its pending nodes. The input rule is first checked for applicability to the sequent corresponding to the pending node. If successful, the rule is attached to the node, then new children nodes are attached according to the outcome of the application of the rule.

Pruning: A proof tree can be pruned at any of its proof tree nodes. The rule and the children associated with that node will be removed (reset to null).

Getting pending nodes: A list of pending nodes can be computed for any proof tree.

Checking completeness: A proof tree is complete if it does not contain any pending proof tree nodes.

Although we do not go into this in detail here, an important property of the proof tree is that the Proof Obligation Manager can calculate the proof dependency using information about used hypotheses and goals at each node of the proof tree. More details on how this is done can be found in [33]. This enables the Proof Obligation Manager to efficiently check for applicability of a proof when the corresponding obligation has changed.

7.5 Tactics

Tactics provide a convenient way to construct and manipulate proofs. The input of a tactic is a proof tree node which will be used as the point of application. A tactic is successful if it modifies the proof tree. The output of a tactic is a Boolean to indicate if it it was successfully applied. For clarification, we categorise tactics into types: *basic tactics* and *tacticals*.

Basic tactics are those that do not depend on other tactics. The following tactics are of this type.

Prune: A tactic that directly use the pruning facility from the Proof Manager. This tactic is successful if the input node is non-pending.

Reasoner Application: Tactics of this class provide a wrapper around a reasoner. The tactic is applied successfully if the reasoner is applicable to the input node. As an example, we have tactics **HypTac** and **ConjGoalTac** corresponding to *Goal in Hypothesis* and *Split Conjunctive Goal* reasoners as described in Section 7.3.1.

Tacticals are tactics that are constructed from other tactics. They usually indicate different strategic or heuristic decisions. In order to construct this type of tactics, the Proof Obligation Manager provides three different operations as follows.

OnAllPending(t): Apply a sub-tactic t to all pending nodes starting from the point of application. This tactical tactic is applied successfully if the sub-tactic t is applied successfully on one of the pending nodes.

Repeat(t): Repeating a sub-tactic t to the point of application until the tactic is not successful. This tactical tactic is applied successfully if the sub-tactic t is applied successfully at least once.

$t_1; \ldots; t_n$: Sequentially composing a list of sub-tactics $t_1, \ldots, t_n$ to apply at the point of application. This tactical tactic is applied successfully if one of the sub-tactic is applied successfully.

More complex proof strategies can be constructed by recursively applying the above operations.

As an example, we can encode a tactic that repeatedly splits conjunctive goals on all pending nodes until no more conjunctive goals exist then try to apply *goal in hypotheses* to discharge pending sub-goals. This tactic can be encoded as follows.

ConjGoalThenHyp ==
Repeat(OnAllPending(**ConjGoal**));
OnAllPending(**Hyp**)

7.6 Automatic and Interactive Modes

As mentioned earlier, the Proof Obligation Manager works both automatically as part of the tool-chain (auto-proving process) and interactively with the Proving User Interface (manual-proving process). At times, the auto-prover may be running in parallel with the manual-prover on the same proof obligation. In order to manage the synchronisation between the two modes, the Proof Obligation Manager uses different working copies of the proofs. For example, the user is currently proving some obligations and decides to make some changes to his model. These changes effect the proof obligations that he is working on. The Proof Obligation Manager should be responsive so that the user does not work on out-of-date obligations. Moreover, the Proof Obligation Manager also should act in a way that the user does not need to close all his proving sessions and re-open them again. In other words, the change in the modeling should be smoothly reflected to the user working on the proving end. This is an important usability consideration of our Rodin toolset. Moreover, the technical detail described in this section is also to ensure that manual proofs can be preserved and reused as long as possible.

For the above reason, the Proof Obligation Manager maintains three different copies of the proofs for each obligation: a copy on disc for persistency (persistent copy - PC), a copy for auto-proving (AC), and a copy for manual-proving (MC). The relationships between these copies can be seen in Figure 4.

The auto-proving and manual-proving process have exclusive access to update the PC. Moreover, the manual-proving process can load and listen to the changes from the PC.

Different statuses are associated with various copies of the proofs. The only common status is the confidence

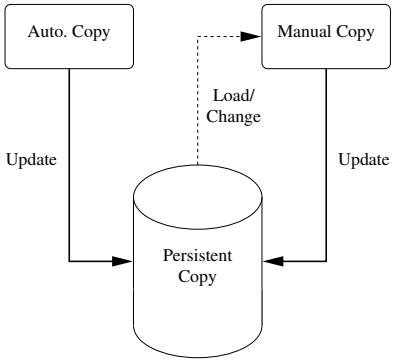


Fig. 4: Different working copies of proof statuses

level of the proof (discharged, reviewed or pending). In particular, for PC, there are two additional statuses:

Source: To indicate the source of the proof, either Automatic or Manual.

Out of Date: A Boolean status to indicate if the proof is out-of-date with the proof obligation.

We distinguish between automatic and manual proofs so that the tool can keep a manually constructed proof even if a proof obligation change has rendered this proof invalid. This allow it to be used later if it becomes valid again.

7.6.1 Auto-Proving Actions

The auto-proving process creates an initial pending node based on the proof obligation and invokes some predefined automatic tactics. Typically this happens as part of the tool-chain when the proof obligation changes. Upon finishing, the auto-proving process updates the PC only if the proof obligation is proved, so that any (old) manual proof is still preserved if the auto-proof fails (even though it might not be valid).

7.6.2 Manual-Proving Actions

The manual-proving process initially loads the proof from the PC as its MC. When updating the PC, the manual-proving process only saves modified proofs and sets the source status to *Manual*. Similar to the auto-proving process, the manual-proving process also tries to preserve the manual proofs as much as possible. Moreover, if the proof obligation has changed the manual-proving process needs to present to the user the up-to-date information in order to help the user avoid working on out-of-date proof obligation. For this, the manual-proving process listens to the changes in the PC (e.g. updated by the auto-proving process) and changes its states accordingly.

There are two different ways that the manual-proving process reconstructs a proof when the corresponding proof obligation changes.

Reuse: Checking if the proof can be reused is efficient based on the information of proof dependency. Reuse of a proof for a new obligation is also straight-forward, again based on the dependency information at each proof node. This reuse process does not require the reasoner to be re-run at each node.

Rebuild: A proof can be "rebuilt" for a new proof obligation by re-trying the reasoner at each proof node. This process is recursively apply to all the children nodes if the reasoner is successful. Otherwise, the rebuilding stop for this node. The idea of rebuilding a proof is to try to carry out the same proof as far as possible.

8 Well-definedness in modelling and proof

This section covers the treatment of partial functions in the Rodin tool. Partial functions are frequently used for modelling in Event-B. For example in Section 4 we used the partial function expression $status(u)$. Using partial functions entails reasoning about potentially ill-defined expressions in proofs which can be tedious and problematic to work with. Providing proper logical and tool support for reasoning in the presence of partial functions is therefore important in our setting.

The Rodin tool provides support for *well-definedness* in order to aid the activities of modelling and proving. By supporting well-definedness we mean that it is ensured that partial functions are never applied to arguments outside their domain. The formal definition of the notion of well-definedness used by the Rodin platform can be found in [32]. A novelty of this approach is that the logic used is an extension of standard predicate calculus. Because of this, all proofs are reducible to standard predicate calculus, which is widely understood and has well-developed automated proof support. Further details and comparisons with other approaches that deal with partial functions can be found in [32]. In this section we will only summarise the user's view of the tool support for well-definedness provided and how it aids working with partial functions.

The tool ensures that partial functions are never applied to arguments outside their domain. This is achieved by *filtering* all mathematically relevant user input entered at the time of modeling or proving to reject any user input containing potentially ill-defined expressions. Figure 5 illustrates how well-definedness can be thought of as an additional proof-based filter for mathematical texts. The treatment of well-definedness within the Rodin platform is done as follows:

During Modeling: In order to ensure that a model or context is well-defined, the proof obligation generator generates well-definedness proof obligations for each modeling element whose well-definedness cannot be trivially guaranteed on the basis of some simple syntactic rules.

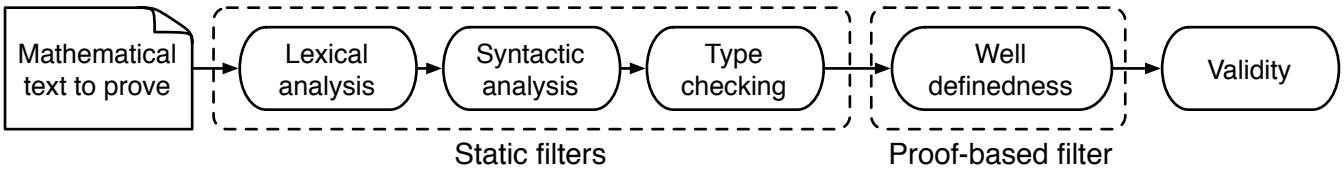


Fig. 5: Well-definedness as an additional filter

For instance, an invariant with no occurrences of the function application symbol is trivially well-defined. An example of a well-definedness proof obligation for the expression $status(u)$ was shown as (4) in Section 4 . Further details on well-definedness of models can be found in [23] and [13].

During Proving: As a result of requiring all models and contexts to be well-defined, all proof obligations presented to a user are also well-defined. This is shown in [23] and [13]. The assumption that a proof obligation is well-defined can be used to greatly ease and shorten its proof (this is shown in §4 of [32]). It is therefore advantageous to preserve well-definedness when carrying out a proof. The inference rules built into the Rodin Prover therefore *preserve* well-definedness. They are similar to standard predicate calculus rules, except that they require the user to additionally prove the well-definedness of all predicates or expressions that they introduce, for instance when adding a lemma, or instantiating a universally quantified hypothesis. As an example, when a user wishes to add the lemma "$3/x = y$" as a hypothesis to a proof, they need to prove not only the lemma itself, but also its well-definedness predicate "$x \neq 0$". Details on how well-definedness predicates are calculated can be found in [32].

Furthermore, since all proofs can be reduced to standard predicate calculus, the user may choose to discharge a subgoal at any time using one of the many freely available external automated theorem provers for predicate calculus, even though they have no support for well-definedness.

The support for well-definedness provided by the Rodin tool, as outlined above, aids the tasks of modeling and proving since:

- It provides the user with quick design-time feedback on possibly erroneous ill-defined expressions during modeling and proof.
- It preserves well-definedness while performing a proof and uses this assumption of well-definedness to shorten and ease proof.
- It still allows external automated theorem provers with no support for well definedness to be used to discharge sub-goals.

Although the task of establishing and preserving well-definedness during modeling and proving increases the total number of proof obligations and sub-goals that that user has to prove, in practice we have noticed that these proofs require only minimal additional effort and can, in most cases, be discharged automatically. The advantages of establishing and preserving well-definedness therefore seem to outweigh their additional cost.

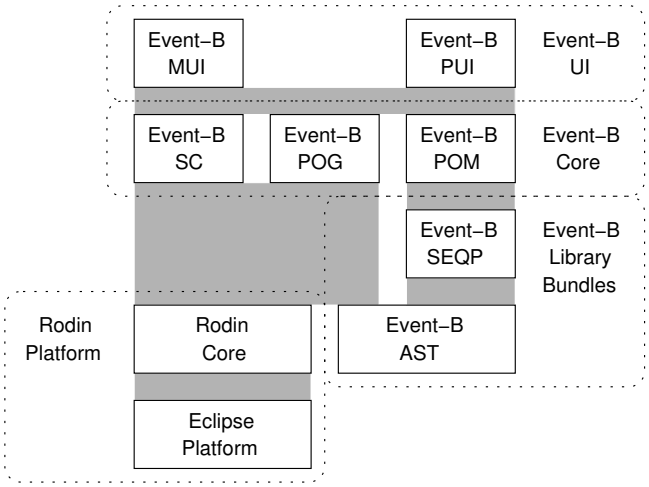


Fig. 6: Architectural Overview of the Event-B Tool

9 Implementation Architecture

The Rodin tool (see Figure 6) is an extension of the Eclipse platform. We do not explain Eclipse in this article but only refer to the existing literature [22]. The tool is implemented in Java though some of the plug-ins described in Section 10 include large parts written in other languages (e.g., Prolog).

9.1 The Rodin Core

The Rodin Core consists of two components: the Rodin repository and the Rodin builder. These two components are tightly integrated into Eclipse based on designs derived from the Java Development Tools of Eclipse. Informal specifications for the repository and the builder have been developed. Their functionality is simple. They are however very dependent on the resources and concurrency model of Eclipse. Neither the repository nor the builder make any assumptions about elements being

stored. In particular, they are independent of Event-B. The use of a repository instead of a fixed syntax for the modelling notations makes extending Event-B much easier. It is not necessary to change the syntax or to make extensions inside comments (in order not to change the syntax).

The Rodin repository manages persistence of data elements. There is a simple correspondence between data elements in form of Java objects and their persistent storage in XML files. The main design characteristic of the Rodin repository is easy extensibility.

The Rodin builder schedules jobs depending on changes made to files contained in the Rodin repository. The builder concept is supplied by the Eclipse platform. It is responsible for automatically launching jobs in the background to achieve higher responsiveness. The builder can be extended by adding new tools to it that keeps derived data elements in the Rodin repository up to date.

9.2 The Event-B Library Packages

The full Event-B language of contexts and machines does not have a concrete syntax that needs to be parsed. Instead Event-B models are maintained in a structured repository. However, the mathematical notation used, e.g., in invariants or guards, does have a concrete syntax. It is specified by an attributed grammar that is used to produce the abstract syntax tree (AST) package. The grammar has not been specified in Event-B, although, in principle this should be possible similarly to the technique proposed by Lamport based on TLA+ [28].

The sequent prover (SEQP) library provides the proof engine. It contains the necessary data types, notably the sequent data type, some inference rules and support for tactics. The inference rules have been chosen to represent proof trees that can be easily manipulated in interactive proofs (see Section 7).

9.3 The Event-B Core

The Event-B Core consists of three components: the static checker (SC), the proof obligation generator (POG), and the proof obligation manager (POM). Their connection is shown in Figure 1. The scheduling of the three components is taken care of by the Rodin builder. The role of the POG was covered in Section 5 and the role of the POM was covered in Section 7.

The Event-B static checker (SC) analyses Event-B contexts and Event-B machines and provides feedback to the user about syntactical and typing errors in them. The mathematical notation of Event-B is specified by a context-free grammar, whereas the rest of Event-B is specified by a graph grammar based on the repository elements. The static checker rejects repository elements that do not satisfy the context-free grammar and produces error messages. It does, however, accept those repository elements that do satisfy the context-free grammar for use by the proof obligation generator. This mechanism supports incremental development by allowing proofs to go ahead for those repository elements of a model that are statically valid. The static checker can be extended by rejecting more elements and by dealing with new elements that can be added to the repository.

The proof obligation generator produces proof obligations that have already been simplified. This makes them easier to prove automatically and to read in case automatic proof fails. The role of the static checker is to filter all elements from the repository that would cause errors in the proof obligation generator. Separating the two yields a much simplified proof obligation generator. This separation is similar to that of front-end and code generator in a compiler.

10 Openness and Extension

We take the view that no one tool can solve all our development problems and that it is important to apply a range of tools in a complementary way in rigorous development. For example, it makes sense to apply model checking as a pre-filter, before applying a theorem prover to a proof obligation. Similarly the use of diagrammatic views (e.g., UML) of a formal model can aid with construction and validation. Many analysis tools, such as model checkers, theorem provers, translation tools (e.g., UML to B and code generators), have been developed, some of which are commercial products and some research tools. However a major drawback of these tools is that they tend to be closed and difficult to use together in an integrated way. They also tend to be difficult for other interested parties to extend, making it difficult for the work of a larger research community to be combined.

Our aim with the Rodin open toolset is to greatly extend the state of the art in formal methods tools, allowing multiple parties to integrate their tools as plug-ins to support rigorous development methods. The open architecture of Rodin allows other parties to integrate their tools, such as model checkers and theorem provers, as plug-ins to support rigorous development. This will allow many researchers to contribute to the provision of a comprehensive integrated toolset and we believe it will encourage greater industrial uptake of these tools.

As well as supporting the combination of different complementary tools, openness and customizability is very important in that it will allow users to customize and adapt the basic tools to their particular needs. For example, a car manufacturer using Event-B to study the overall design of a car information system might be willing to plug some special tools able to help defining the corresponding documentation and maintenance package. Likewise, a rocket manufacturer using Event-B might be willing to plug a special tool for analysing and developing the failure detection part of its design.

We outline two significant plug-in tools that have been developed for Rodin, ProB and UML-B. These plug-ins provide valuable additional functionality that complement the existing modelling and proof functionality of the core Rodin platform.

10.1 ProB

The ProB animator and model checker has been presented in [30]. Based on Prolog, the ProB tool supports automated consistency checking of B machines via *model checking*. For exhaustive model checking, the given sets must be restricted to small finite sets, and integer variables must be restricted to small numeric ranges. This allows the checking to traverse all the reachable states of the machine. ProB can also be used to explore the state space non-exhaustively and find potential problems. The user can set an upper bound on the number of states to be traversed or can interrupt the checking at any stage. ProB will generate and graphically display counter-examples when it discovers a violation of the invariant. ProB can also be used as an animator of a B specification. So, the model checking facilities are still useful for infinite state machines, not as a verification tool, but as a sophisticated debugging and testing tool. ProB is available as a plug-in for Rodin.

10.2 UML-B

The UML-B [38] is a profile of UML that defines a formal modelling notation. It has a mapping to the Event-B language. UML-B consists of class diagrams with attached statecharts, and an integrated constraint and action language, called μB, based on the Event-B notation. UML-B provides a diagrammatic, formal modelling notation based on UML. The popularity of the UML enables UML-B to overcome some of the barriers to the acceptance of formal methods in industry. Its familiar diagrammatic notations make specifications accessible to domain experts who may not be familiar with formal notations.

The UML-B [38] plug-in converts UML-B models into Event-B models. Translation from UML-B into Event-B enables the Rodin proof obligation generator and provers to be utilised. Since the Event-B language is not object-oriented, class instances must be modelled explicitly in the generated Event-B. Attributes and associations are represented as variables whose type is a function from the class instances to the attribute type or associated class. Operation behaviour may be represented textually in μB, as a state chart attached to the class, or as a simultaneous combination of both. Further details of UML-B are given in [38].

11 Roadmap

In its present form, Rodin provides a powerful and effective toolset for Event-B development and it has been validated by means of numerous medium-sized case studies. Naturally further improvements and extensions are required in order to improve the productivity of users further and in order to scale the application of the toolset to large industrial-scale developments. We outline the main extensions to Rodin that we have planned for a four year time frame.

11.1 Scaling

Composition and decomposition: Composition and decomposition of models is essential for scalability. There are plans to support two styles of composition for Event-B in Rodin:

Style A Sub-models interact via shared variables
Style B Sub-models interact via synchronisation over events

Rodin will be extended to provide support for composing models as well as decomposing models according to these styles. The proof obligation generator will be extended to enable independent refinement of sub-models.

Team-based development: Support for composition and decomposition will go some way towards enabling team-based development. But there will still be situations where a team needs to access a common set of models. Rodin will be extended to support concurrent modification of developments by providing viewing of change conflicts and automated merge of changes. It will provide support for version control. Support to analyse the impact of multiple user modifications on proof will be investigated.

11.2 Extending the proof obligations and theory:

Proof obligations: Event-B models will be extended to include external variables. The proof obligation for such variables is that they must be preserved via a functional gluing invariant between abstract and concrete external variables. Other forms of proof obligations will also be added to support different paradigms (concurrent, distributed, sequential systems). These include proof obligations for preservation of event enabledness and richer variant structures (such as pointwise ordering and lexicographic ordering) for convergence proof obligations.

Mathematical extensions: Rodin will be extended to support richer types such as record structures and user-defined data types including inductive data types. Appropriate automated and interactive proof support for richer types will be investigated and provided. Higher order provers should enable proof support for inductive datatypes. Users will be able to define operators of polymorphic type (but not use operator overloading) as well

as parameterised predicate definitions. Support for disjointness constraints will be added.

Proof and model checking: Rodin provides an open architecture for proof in the form of a proof manager that can use a range of provers to discharge proofs and sub-proofs. The existing automated provers will be extended with more powerful decision procedures. The use of existing first order and higher order automated provers will be investigated. As mentioned already, higher order provers should enable proof support for inductive datatypes. The possibility of exploiting automated techniques such as SMT [12] and SAT [25] will be investigated.

Animation: Prototype animation plug-ins already exist. The animation facilities will be extended to allow for greater automation of large animations to support regression testing of models. A clear API to the animation will be provided to allow for easy integration with graphical animation tools.

11.3 Process and productivity

Requirements Handling and Traceability: The interplay between informal requirements and formal modelling is crucial in system development and needs better tool support. Facilities for constructing structured requirements documents and for building links between informal and formal elements will be added to Rodin. These will support traceability between requirements and formal models. Support for recording validation of these links and for managing consistency under change to requirements and to formal models will be provided.

Document management: Currently, the B2Latex plug-in for Rodin generates a LaTeX version of an Event-B model. The structure of the document follows the structure of the model. For proper document generation tool support will be provided whereby users dictate the order in which parts of the model are presented. They should be able to write a document, structured according to their needs that includes parts of an Event-B project and that is automatically kept in synchrony with the models.

Automated model generation: Automatic generation of refinements will be investigated and appropriate tool support provided. More general modelling and refinement patterns, enabling greater reuse of modelling and refinement idioms, will be investigated and tool support provided. Code generation from models will be investigated. An indirect route for achieving code generation will be to generated classical B and use the existing code generators for classical B.

12 Conclusion

The Rodin tool is intended to offer the same reactive environment for constructing and analysing models as do modern integrated development environments for programming.

We believe that modelling will remain difficult. This does not mean, however, that it is impossible to develop a productive modelling tool. Programming is difficult, too. Still we have very efficient programming tools. But we also have many people who simply got used to the difficulties of programming. Hopefully, they will also get used to the difficulties of modelling when appropriate tools are available.

The Rodin tool provides a seamless integration between modelling and proving. This is important for the user to focus on the modelling task and not on switching between different tools. The purpose of modelling is not just to write a specification. It also serves to improve our understanding of the system being modelled. The Event-B tool tries to reflect this view by providing a lot of help for exploring a model and reasoning about it.

The tool is extensible and configurable because we cannot predict future uses of Event-B. The architecture has been designed to make this as easy as possible to invite users who need a (formal) modelling tool tailor it to their needs. We hope this will make it possible to employ the tool in very different development processes.

Acknowledgements: We would like to thank the many contributors to the Rodin platform and plug-ins including Nicolas Beauger, Jens Bendisposto, Mathieu Clabaut, Kriangsak Damchoom, Andy Edmunds, Fabian Fritz, Andreas Fürst, Alexei Iliasov, Michael Jastram, Thierry Lecomte, Michael Leuschel, Issam Maamria, Christophe Metayer, Carine Pascal, Antoine Requet, Abdolbaghi Rezazadeh, Mar Yah Said, Matthias Schmalz, Renato Silva, Colin Snook, Francois Terrier. In addition, Dominique Cansell and Cliff Jones provided valuable feedback on the design of the tool. We also thank the anonymous referees for helping us to improve the paper.

References

1. Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
2. Jean-Raymond Abrial. Modelling in Event-B: System and software design. To be published by Cambridge University Press, 2010.
3. Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, and Laurent Voisin. An open extensible tool environment for Event-B. In Z. Liu and J. He, editors, *ICFEM 2006*, volume 4260, pages 588–605. Springer, 2006.
4. Jean-Raymond Abrial and Dominique Cansell. Click'n'Prove: Interactive Proofs within Set Theory. In *Theorem Proving in Higher Order Logics*, volume 2758 of *LNCS*, pages 1–24, 2003.
5. Jean-Raymond Abrial and Stefan Hallerstede. Refinement, decomposition, and instantiation of discrete models: Application to Event-B. *Fundam. Inform.*, 77(1-2):1–28, 2007.
6. R. J. R. Back. Refinement calculus, part II: Parallel and reactive programs. In J. W. de Bakker, W.-P. de

Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems*, volume 430 of *Lecture Notes in Computer Science*, pages 67–93. Springer-Verlag, 1990.
7. R. J. R. Back and J. von Wright. Refinement calculus, part I: Sequential nondeterministic programs. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems*, volume 430 of *Lecture Notes in Computer Science*, pages 42–66. Springer-Verlag, 1990.
8. Ralph-Johan Back. Refinement Calculus II: Parallel and Reactive Programs. In J. W. deBakker, W. P. deRoever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems*, volume 430 of *Lecture Notes in Computer Science*, pages 67–93. Springer, May 1989.
9. Frédéric Badeau and Arnaud Amelot. Using B as a high level programming language in an industrial project: Roissy VAL. In Helen Treharne, Steve King, Martin Henson, and Steve Schneider, editors, *ZB 2005*, volume 3455 of *LNCS*, pages 334–354, 2005.
10. M. Balser, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums. Formal system development with KIV. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering*, number 1783 in LNCS. Springer, 2000.
11. Mike Barnett, Bor-Yuh Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *FMCO 2005*, volume LNCS. Springer-Verlag, 2005. to appear.
12. Clark Barrett, Silvio Ranise, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2008.
13. Patrick Behm, Lilian Burdy, and Jean-Marc Meynadier. Well defined B. In *B '98*, pages 29–45, London, UK, 1998. Springer-Verlag.
14. Yves Bertot and P. (Pierre) Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Texts in theoretical computer science. Springer-Verlag, 2004.
15. Egon Börger and Robert Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.
16. Achim D. Brucker, Frank Rittinger, and Burkhart Wolff. HOL-Z 2.0: A proof environment for Z-specifications. *Journal of Universal Computer Science*, 9(2):152–172, February 2003.
17. Michael J. Butler. Stepwise refinement of communicating systems. *Science of Computer Programming*, 27(2):139–173, 1996.
18. Clearsy. Atelier B tool homepage. http://www.atelierb.societe.com/.
19. David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.
20. Eclipse. Eclipse platform homepage. http://www.eclipse.org/.
21. J.-C. Filliâtre. Verification of Non-Functional Programs using Interpretations in Type Theory. *Journal of Functional Programming*, 13(4):709–745, July 2003.
22. Erich Gamma and Kent Beck. *Contributing to Eclipse*. Addison Wesley, 2003.
23. Stefan Hallerstede. The Event-B Proof Obligation Generator. Technical report, ETH Zürich, 2005.
24. Stefan Hallerstede. Justifications for the Event-B Modelling Notation. In J. Julliand and O. Kouchnarenko, editors, *B 2007*, volume 4355 of *LNCS*, pages 49–63. Springer, 2007.
25. Holger H. Hoos and Thomas Stützle. SATLIB: An Online Resource for Research on SAT. In I.P. Gent, H.V. Maaren, and T. Walsh, editors, *SAT 2000*, pages 283–292. IOS Press, 2000.
26. Matt Kaufmann and J. Strother Moore. An industrial strength theorem prover for a logic based on common lisp. *IEEE Transactions on Software Engineering*, 23(4):203–213, 1997.
27. James C. King. A new approach to program testing. In *Proceedings of the international conference on Reliable software*, pages 228–233, New York, NY, USA, 1975. ACM Press.
28. Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.
29. Thomas Långbacka and Joakim von Wright. Refining reactive systems in HOL using action systems. In Elsa L. Gunter and Amy P. Felty, editors, *Theorem Proving in Higher Order Logics, 10th International Conference, TPHOLs'97*, volume 1275 of *Lecture Notes in Computer Science*, pages 183–197. Springer, 1997.
30. M. Leuschel and M. Butler. ProB: A Model Checker for B. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *Proceedings FME 2003, Pisa, Italy*, LNCS 2805, pages 855–874. Springer, 2003.
31. Farhad Mehta. Supporting proof in a reactive development environment. In *SEFM*, pages 103–112. IEEE Computer Society, 2007.
32. Farhad Mehta. A practical approach to partiality - a proof based approach. In Shaoying Liu, T. S. E. Maibaum, and Keijiro Araki, editors, *ICFEM*, volume 5256 of *Lecture Notes in Computer Science*, pages 238–257. Springer, 2008.
33. Farhad Mehta. *Proofs for the Working Engineer*. PhD thesis, ETH Zurich, 2008.
34. Carroll Morgan, Thai Son Hoang, and Jean-Raymond Abrial. The challenge of probabilistic Event-B – extended abstract. In Helen Treharne, Steve King, Martin C. Henson, and Steve A. Schneider, editors, *ZB 2005: Formal Specification and Development in Z and B, 4th International Conference of B and Z Users*, volume 3455 of *Lecture Notes in Computer Science*, pages 162–171. Springer, 2005.
35. Tobias Nipkow. Structured Proofs in Isar/HOL. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs (TYPES 2002)*, volume 2646 of *LNCS*, pages 259–278. Springer, 2003.
36. Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer Verlag, 1994.
37. Mark Saaltink. The Z/EVES system. In Jonathan P. Bowen, Michael G. Hinchey, and David Till, editors, *ZUM '97: The Z Formal Specification Notation, 10th International Conference of Z Users*, volume 1212 of *Lecture Notes in Computer Science*, pages 72–85. Springer, 1997.
38. Colin F. Snook and Michael J. Butler. UML-B: Formal modeling and design aided by UML. *ACM Trans. Softw. Eng. Methodol.*, 15(1):92–122, 2006.

39. J. Michael Spivey. *The Z Notation: A Reference Manual*. International Series in Computer Science. Prentice-Hall, New York, NY, second edition, 1992.
40. Daniel Winterstein, David Aspinall, and Christoph Lüth. Proof general / eclipse: A generic interface for interactive proof. In *IJCAI*, pages 1587–1588, 2005.